用户手册

1.0

（ Ruby 版 ）

潍坊高远速通软件有限公司

目录

第一章 介绍 1

本章提要 1
欢迎使用 Hprose 2
安装 Hprose for Ruby 2
客户端需求 2
服务器需求 2
安装方法 3
体例 3
菜单描述 3
屏幕截图 3
代码范例 3
运行结果 3
获取其它帮助信息 4
电子文档 4
在线支持 4
案例网站 4
升级 4
联系我们 4
报告漏洞与需求 4

第二章 快速入门 5

本章提要 5
创建 Hprose 的 Hello 服务器 6
创建 Hprose 的 Hello 客户端 6

第三章 Hprose for Ruby 类型映射 8

本章提要 8
基本类型 9
值类型 9
引用类型 9
基本类型的映射 10
序列化类型映射 10
反序列化类型映射 10
容器类型 11
列表类型 11
字典类型 11
对象类型 11

第四章 Hprose for Ruby 服务器 13

本章提要 13
服务发布方式 14
Rackup 方式 14

独立服务器方式 14
Rails on Rack 方式 15
发布服务 17
发布函数 17
发布语句块 18
发布方法 18
别名机制 19
发布对象 20
发布类 20
迷失的方法 21
服务器开关 21
隐藏发布列表 21
调试开关 22
P3P 开关 22
跨域开关 22
服务器事件 22
on_before_invoke 事件 22
on_after_invoke 事件 23
on_send_header 事件 23
on_send_error 事件 23
会话管理 23
Rack 中间件 23
第五章 Hprose for Ruby 客户端 25
本章提要 25
同步调用 26
直接通过远程方法名进行远程调用 26
通过代理对象进行远程调用 27
通过 Invoke 方法进行远程调用 28
非引用参数传递 28
引用参数传递 28
异步调用 29
非引用参数传递 29
引用参数传递 30
异常处理 30
同步调用中的异常处理 30
异步调用中的异常处理 31
HTTP 参数设置 32
代理服务器 32
HTTP 标头 32

第一章 介绍

在开始用 Hprose for Ruby 开发应用程序前，您需要先了解一些相关信息。阅读这一章能让您知道如何安装 Hprose for Ruby 和如何得到帮助。

本章提要

- 欢迎使用 Hprose
- 安装 Hprose for Ruby
- 体例
- 获取其它帮助信息
- 联系我们

欢迎使用 Hprose

Hprose 使您能高效便捷的创建功能强大的跨平台、跨语言、分布式的应用。如果您刚接触网络编程，您会发现用 Hprose 来实现分布式系统易学易用。如果您是一位有经验的程序员，您会发现它是一个功能强大的通讯协议和开发包。有了它，您在任何情况下，都能在更短的时间内完成更多的工作。

Hprose 是 PHPRPC 的商业化版本，它除了拥有 PHPRPC 的各种优点之外，它还具有更多自己的特别之处。Hprose 使用自己独特的方式表示数据，在更加节省空间的同时，可以表示更多的数据类型，解析效率也更加高效。在数据传输上，Hprose 以更直接的方式来传输数据，不再需要二次编码，可以直接进行流式读写，效率更高。在远程调用过程中，数据直接被还原为目标类型，不再需要类型转换，效率上再次得到提高。Hprose 不仅具有在 HTTP 协议之上工作的版本，以后还会推出直接在 TCP 协议之上工作的版本。Hprose 在易用性方面也有很大的进步，您几乎不需要花什么时间就能立刻掌握它。

Hprose 与其它远程调用商业产品的区别很明显，那就是 Hprose 是开源的，您可以在购买相应的授权以后即可获得源代码的阅读或修改权，这样您就可以在遇到问题时更快的找到问题并修复它，或者在您无法直接修复的情况下，更准确的将错误描述给我们，由我们来帮您更快的解决它。您还可以将您所修改的更加完美的代码或者由您所增加的某个激动人心的功能反馈给我们，让我们能够更好的来一起完善它。正是因为有这种机制的存在，您在使用该产品时，实际上可能遇到的问题会更少，因为问题可能已经被他人修复了。

Hprose 与其它远程调用开源产品的区别更加明显，Hprose 不仅仅在开发运行效率，易用性，跨平台和跨语言的能力上较其它开源产品有着明显的不可取代的综合优势，Hprose 还可以保证所有语言的实现具有一致性，而不会向其他开源产品那样即使是同一个通讯协议的不同实现都无法保证良好的互通。而且 Hprose 具有完善的商业支持，可以在任何时候为您提供所需的帮助。而不会向其它没有商业支持的开源软件那样，当您遇到问题时只能通过阅读天书般的源代码的方式来解决。

Hprose 支持许多种语言，包括您所常用或不常用甚至从来不用的语言。您不需要掌握 Hprose 支持的所有语言，您只需要掌握您所使用的语言就可以开始启程了。

本手册是针对 Hprose 的 Ruby 版本来展开讲解的，其中有些内容您可能在其它语言版本的手册中也会看到，我们之所以会在不同语言的手册中重复这些内容是因为我们希望您只需要一本手册就可以掌握 Hprose 在这种语言下的使用，而不需要同时翻阅几本书才能有一个全面的认识。

下面我们先来看看如何安装 Hprose for Ruby 吧。

安装 Hprose for Ruby

Hprose for Ruby 对硬件和操作系统需求如下：

客户端需求

任何安装有 Ruby 1.8.5 及其更高版本的操作系统和硬件。

服务器需求

任何安装有 Ruby 1.8.5 的操作系统和硬件。

服务器端可以不依赖于 Web 服务器独立运行，也可以配合任何支持 Rack 的 Web 服务器运行。

安装方法

如果您已经安装了 Ruby，进入 hprose for ruby 的目录，如果其中没有 hprose-1.0.0.gem 的话，您可以执行：

```
ruby gem.spec
```

生成 hprose-1.0.0.gem。

然后在该目录下执行：

```
gem install hprose
```

即可安装成功。

体例

菜单描述

当让您选取菜单项时，菜单的名称将显示在最前面，接着是一个箭头，然后是菜单项的名称和快捷键。例如“文件→退出”意思是“选择文件菜单的退出命令”。

屏幕截图

Hprose 是跨平台的，支持多个操作系统下的多个开发环境，因此文档中可能混合有多个系统上的截图。

代码范例

代码范例将被放在细边框的方框中：

```
#!/usr/bin/ruby
def hello(name)
  'Hello ' + name + '!'
end
puts hello('World')
```

运行结果

运行结果将被放在粗边框的方框中：

```
Hello World!
```

获取其它帮助信息

电子文档

只要您拥有用户帐号，所有的 Hprose 文档都能从我们的网站（ http://www.hprfc.com ）和 FTP（ ftp.hprfc.com ）上下载。这些文档都是 PDF 格式的。您也可以向我们购买这些电子文档的印刷版。

在线支持

我们的技术支持网页为 http://www.hprfc.com/support。该页面是查询有关 Hprose 信息的地方。您会找到客户服务、技术服务方面的信息以及官方用户论坛、邮件列表和新闻组的链接。

案例网站

现在有数十个 Hprose 用户创建的提供开发技术方面信息的网站，您可以通过我们网站上与之有关的链接来访问它们。

升级

我们每三个月到半年发布一个 Hprose 的新版本。凡是有升级许可证的用户都能升级。

联系我们

如果您需要直接跟我们取得联系，可以使用下列方法：

公司名称	潍坊高远速通软件有限公司
公司地址	潍坊市高新区玉清东街以北中心次干道以西高新大厦（ 邮编 261061 ）
电子邮件	市场及大型项目合作： manager@hprfc.com 产品购买及项目定制： sales@hprfc.com 技术支持： support@hprfc.com
联系电话	+86-0536-2999015（ 周一至周五，北京时间早上 9 点到下午 5 点 ）

报告漏洞与需求

如果您在 Hprose 中发现了漏洞或需要一些新的功能，请告知我们。最好的方法是通过我们公司网站的反馈系统进行反馈。我们的反馈系统专门用于收集所有有用的信息，它能帮助我们捕获缺陷和了解用户对新功能的需求。您每报告一个缺陷或功能需求后，都将会相应的收到一封电子邮件，其中含有跟踪号码，通过它，您可以及时了解报告的处理状态。当我们完成对报告的处理，我们会发送电子邮件告知您处理结果及原因（ 例如：缺陷将在下次发布中被解决、 新特性将在下次发布时加入进来这并不是缺陷等等 ）。

第二章 快速入门

使用 Hprose 制作一个简单的分布式应用程序只需要几分钟的时间，您先要创建服务器端应用程序，然后创建客户端应用程序，最后您就可以将它们运行起来了。本章将用一个简单但完整的实例来带您快速浏览使用 Hprose for Ruby 进行分布式程序开发的全过程。

本章提要

- 创建 Hprose 的 Hello 服务器
- 创建 Hprose 的 Hello 客户端

创建 Hprose 的 Hello 服务器

创建 Ruby 的 Hprose 的服务器非常简单。Hprose for Ruby 服务器是标准的 Rack 应用程序，所以你需要先安装 rack，安装很简单，只需要执行：

```
gem install rack
```

就安装成功了。

然后创建服务发布程序，你只需几行代码就可以创建它：

```
require "rubygems"
require "hproseserver"
def hello(name)
  'Hello ' + name + '!'
end
app = HproseHttpService.new()
app.add(method(:hello))
run app
```

将上面这段代码保存为 helloservice.ru，注意扩展名是 ru，不是 rb。

然后执行：

```
rackup helloservice.ru
```

现在你可以在同一台机器上打开浏览器，输入：http://127.0.0.1:9292/，然后回车，看到浏览器中有如下输出：

```
Fa1{s5"hello"}z
```

就表示我们的服务发布成功啦。接下来我们来看一下客户端如何创建吧。

创建 Hprose 的 Hello 客户端

Ruby 的 Hprose 客户端创建更加容易，下面我们来创建一个 helloclient.rb，内容如下：

```
#!/usr/bin/ruby
require "rubygems"
require "hproseclient"
client = HproseHttpClient.new('http://localhost:9292/')
puts client.hello('World')
```

然后在命令行中输入：

```
ruby helloclient.rb
```

如果看到下面的结果：

```
Hello World!
```

就说明客户端创建成功了！

请注意，上面运行客户端之前必须要先把服务器运行起来，否则你可能会看到类似如下的错误输出：

```
C:/Ruby/lib/ruby/1.8/net/http.rb:560:in `initialize': 由于目标计算机积极拒绝，无法连接。 - connect(2)
(Errno::ECONNREFUSED)
        from C:/Ruby/lib/ruby/1.8/net/http.rb:560:in `open'
        from C:/Ruby/lib/ruby/1.8/net/http.rb:560:in `connect'
        from C:/Ruby/lib/ruby/1.8/timeout.rb:53:in `timeout'
        from C:/Ruby/lib/ruby/1.8/timeout.rb:93:in `timeout'
        from C:/Ruby/lib/ruby/1.8/net/http.rb:560:in `connect'
        from C:/Ruby/lib/ruby/1.8/net/http.rb:553:in `do_start'
        from C:/Ruby/lib/ruby/1.8/net/http.rb:548:in `start'
        from C:/Ruby/lib/ruby/gems/1.8/gems/hprose-1.0.0/lib/hprose/httpclient.rb:84:in `_post'
        from C:/Ruby/lib/ruby/gems/1.8/gems/hprose-1.0.0/lib/hprose/httpclient.rb:70:in `send_data'
        from C:/Ruby/lib/ruby/gems/1.8/gems/hprose-1.0.0/lib/hprose/client.rb:99:in `_invoke'
        from C:/Ruby/lib/ruby/gems/1.8/gems/hprose-1.0.0/lib/hprose/client.rb:67:in `invoke'
        from C:/Ruby/lib/ruby/gems/1.8/gems/hprose-1.0.0/lib/hprose/client.rb:121:in `method_missing'
        from helloclient.rb:4
```

到这里，您应该已经掌握 Hprose for Ruby 的基本用法啦。接下来，就让我们一起对 Hprose for Ruby 进行深层探秘吧。

第三章 Hprose for Ruby 类型映射

类型映射是 Hprose 的基础，正是因为 Hprose 设计有良好的类型映射机制，才使得多语言互通得以实现。本章将对 Hprose for Ruby 的类型映射进行一个详细的介绍。

本章提要

- 基本类型
- 容器类型
- 对象类型

基本类型

值类型

类型	描述
整型	Hprose 中的整型为 32 位有符号整型数，表示范围是-2147483648～2147483647（-2^{31}～$2^{31}-1$）。
长整型	Hprose 中的长整型为有符号无限长整型数，表示范围仅跟内存容量有关。
浮点型	Hprose 中的浮点型为双精度浮点型数。
非数	Hprose 中的非数表示浮点型数中的非数（NaN）。
无穷大	Hprose 中的无穷大表示浮点型数中的正负无穷大数。
布尔型	Hprose 中的布尔型只有真假两个值。
字符	Hprose 中的 UTF8 编码的字符，仅支持单字元字符。
空	Hprose 中的空表示引用类型的值为空（null）。
空串	Hprose 中的空串表示空字符串或零长度的二进制型。

其中非数和无穷大其实是特殊的浮点型数据，只不过在 Hprose 中它们有单独的表示方式，这样可以使它们占用更少的存储空间，并得到更快的解析。

另一个可能会引起您注意的是，这里把空和空串也作为值类型对待了。这里把它列为值类型而不是引用类型，是因为 Hprose 中的值类型和引用类型的概念与程序设计语言中的概念不完全相同。这里的值类型是表示在 Hprose 序列化过程中，不做引用计数的类型。在序列化过程中，当遇到相等的值类型时，后写入的值将与先写入的值保持相同的形式，而不是以引用的形式写入。

引用类型

类型	描述
二进制型	Hprose 中的二进制型表示二进制数据，例如字节数组或二进制字符串。
字符串型	Hprose 中的字符串型表示 Unicode 字符串数据，以标准 UTF-8 编码存储。
日期型	Hprose 中的日期型表示年、月、日，年份范围是 0～9999。
时间型	Hprose 中的时间型表示时、分、秒（毫秒，微秒，毫微秒为可选部分）。
日期时间型	Hprose 中的日期时间型表示某天的某个时刻，可表示本地或 UTC 时间。

空字符串和零长度的二进制型并不总是表示为空串类型，在某些情况下它们也表示为各自的引用类型。空串类型只是对二进制型和字符串型的特殊情况的一种优化表示。

引用类型在 Hprose 中有引用计数，在序列化过程中，当遇到相等的引用类型时，后写入的值是先前写入的值的引用编号。后面介绍的容器类型和对象类型也都属于引用类型。

基本类型的映射

Ruby 类型与 Hprose 类型的映射关系不是一一对应的。在序列化和反序列化过程中可能会有一种 Ruby 类型对应多种 Hprose 类型的情况出现（当然条件会有不同）。我们下面以列表的形式来说明。

序列化类型映射

Ruby 类型	Hprose 类型
Fixnum 中-2147483648～2147483647 范围之内的数字（含边界）	整型
Fixnum 中-2147483648～2147483647 范围之外的数字，Bignum	长整型
Float	浮点型
0.0/0.0	非数
1.0/0.0	正无穷大
-1.0/0.0	负无穷大
true	布尔真
false	布尔假
nil	空
非 utf8 编码的 String	二进制型（或空串）
utf8 编码的单字元 String	字符
Symbol，utf8 编码的多字元 String	字符串型（或空串）
Time	日期/时间/日期时间型

反序列化类型映射

Hprose 类型	Ruby 类型
整型	Fixnum
长整型	Fixnum，Bignum
浮点型	Float
非数	0.0/0.0
正无穷大	1.0/0.0
负无穷大	-1.0/0.0
布尔真	true

Hprose 类型	Ruby 类型
布尔假	false
空	nil
空串	""
二进制/字符/字符串型	String
日期/时间/日期时间型	Time

容器类型

Hprose 中的容器类型包括列表类型和字典类型两种。它们分别对应于 Ruby 的 Array 和 Hash 类型。

列表类型

Ruby 中的 Array 类型，被映射为 Hprose 列表类型。例如：

```
a = [1, 2, 3, 4, 5]
fruit = ['apple', 'banana', 'cherry']
```

都被序列化为 Hprose 的列表类型。

另外，Range、MatchData 类型也被序列化为 Hprose 的列表类型。

但是反序列化时，Hprose 的列表类型只被反序列化为 Ruby 的 Array 类型。

字典类型

Ruby 中的 Hash 类型被映射为 Hprose 字典类型，例如：

```
a = { 1 => 'one', 2 => 'two', 3 => 'three' }
info = { 'version' => 4,
              'OS' => 'Linux',
            'lang' => 'English',
      'short_tags' => true }
```

都被映射为 Hprose 字典类型。

对象类型

Ruby 中自定义类的对象实例在序列化时被映射为 Hprose 对象类型。自定义类中的属性名，映射为 Hprose 对象类型中的属性名，自定义类中的属性值，映射为 Hprose 对象类型中的属性值，类中所有的属性值必须为上述可序列化类型。

例如：

```
class User
  attr_accessor :name, :age, :birthday, :sex, :married
end
user = User.new()
user.name = 'Tom'
user.age = 28
user.birthday = Time.local(1982, 2, 23)
user.sex = 1
user.married = true
```

当反序列化一个对象时，如果该对象所在的类未定义，该类将会被自动定义。

类可以嵌套定义，嵌套定义的所有类名一起构成类的全名，在类的全名中，用于分割类名的双冒号分隔符会被替换为下滑线分隔符。Hprose 在多语言交互时，对象类型是通过全名来识别的。

例如：

```
class My
  class Package
    class User
      attr_accessor :name, :age, :birthday, :sex, :married
    end
  end
end
```

这个 My::Package::User 类，当我们在跟 PHP 交互时，它是与 PHP 中的 My_Package_User 类对应的。

所以上面的类也可以这样定义：

```
class My_Package_User
  attr_accessor :name, :age, :birthday, :sex, :married
end
```

效果是相同的。

而当跟 java、ActionScript 或 C#这类支持包或名字空间管理的语言进行交互时，它就跟 My.Package.User，My_Package.User，My.Package_User 或者 My_Package_User 这四个类相对应了。

所以我们应该保证这四个类只有一个被定义。如果有多个被定义，结果也只会跟其中某一个建立对应关系，至于是哪一个，这与查找方式有关，不是固定的。所以，应该避免同时定义多个这样的类。

第四章 Hprose for Ruby 服务器

前面我们在快速入门一章里学习了如何创建一个简单的 Hprose for Ruby 服务器，在本章中您将深入的了解 Hprose for Ruby 服务器的更多细节。

本章提要

- 服务发布方式
- 发布服务
- 服务器开关
- 服务器事件
- 会话管理
- Rack 中间件

服务发布方式

Hprose for Ruby 服务从本质上来讲是一个 Rack 应用程序，因此所有 Rack 支持的方式，它都适用。Hprose for Ruby 服务之所有以 Rack 方式来实现有两个原因：

首先，Rack 是 Ruby 在 Web 开发中的一个标准，它与 WSGI 在 Python 中的地位和作用是相当的。因此，采用这种方式来实现，将可以最大程度的支持最多的服务器。

其次，采用 Rack 方式在运行效率上相当高效，因此它可以更充分的发挥服务器的性能。

下面我们来介绍几种最主要的服务发布方式。

Rackup 方式

在前面快速入门一章里，我们介绍了如何以 Rackup 方式来发布服务，我们使用的是默认的 Webrick 服务器，但是在实际部署服务时，我们几乎从来不用这个服务器，这个服务器一般仅用于开发期间的调试。

另外常用服务器还有 mongrel，thin 等。这些服务器的启动方式基本上是相同的，都可以通过 rackup 来启动，例如：

```
rackup -s mongrel helloservice.ru
```

不过前提是，你要安装有这些服务器，安装这些服务器很简单，例如安装 mongrel 只需要执行：

```
gem install mongrel
```

就可以啦。

不推荐在 Windows 下使用 thin，它在 Windows 下运行 rack 应用很不稳定。但是在 Linux 下，thin 表现的很好，因此如果您使用 Linux 作为服务器，则推荐使用 thin。

rackup 有许多参数，可以通过这些参数来设置采用何种服务器，以及监听的主机名、端口号，服务运行模式等许多选项。

独立服务器方式

rackup 方式的好处是，一个 ru 文件可以通过多种服务器来启动它，只要在使用 rackup 时指定不同的服务器参数即可。但是有时候我们就是确定只用某一个服务器，这时候你可以直接以 rb 文件方式来写一个独立运行的服务器，例如下面这个例子：

```
require "rubygems"
require "ebb"
require "hproseserver"
def hello(name)
  'Hello ' + name + '!'
end
app = HproseHttpService.new()
app.add(:hello)
Ebb.start_server(app)
```

它使用了 Ebb 作为发布服务器。Ebb 是 Linux 下使用的一个高性能的 rack 服务器。在 Linux 下可以直接通过 gem install ebb 来安装它，但如果你使用的是 ubuntu，需要注意一定要先用 apt-get 安装好 ruby-dev 这个包。

之后直接执行该程序即可启动服务器了。

独立服务器方式与 rackup 方式有一点区别，例如上面发布的 hello 方法，在独立服务器中，它是属于顶级对象的，也可以说它相当于一个函数，因此可以直接通过 add 方法后面跟方法名:hello（或者字符串 "hello"）。而在 rackup 方式下，在 ru 文件中定义的 hello 方法实际上属于 Rack::Builder 的一个实例，因此如果直接在 add 方法后面跟方法名，将无法发布它，因此只能通过 method 来获取到 hello 方法然后再用 add 来发布它。

独立服务器方式是速度最快的，如果您追求的是速度的极致，那么您可以采用这种方式。

独立服务器的代码需要为不同的服务器单独编写，虽然主要语句没有差别，但在服务器启动的语句上还是有少许差别的，因此这种方式在实际应用中并不多见。

更多被广泛应用的是下面这种 Rails on Rack 方式。

Rails on Rack 方式

Ruby 能有今天这么火热，大半归功于 Rails 的成功。因此大部分的 Ruby 应用实际上都是 Rails 的应用，所以如果能在 Rails 上直接发布 Hprose 服务，就再好不过了。

要实现这个愿望是没有问题，在 Rails 2.3 之后，Rails 也实现了对 Rack 的支持。通过创建 Rails Metal 应用程序，即可实现对 Hprose 服务的发布。

虽然采用 Rails on Rack 方式发布 Hprose 服务的效率赶不上单独的 Rackup 和独立服务器方式。但因为 Rails Metal 应用程序要比普通的 Rails Action 快很多倍，因此在 Rails 上发布 Hprose 服务，比普通的 Rails 应用还是高效的多，因为目前大部分的 Rails 应用还是基于 Rails Action 来实现的。

下面我们来详细看一下如何在 Rails 上发布一个 Hprose 服务。

首先第一步是安装 Rails，通过 gem 很容易安装，在 Windows 命令行或 Linux/Unix 终端上键入：

```
gem install rails
```

然后回车，即可完成安装。

然后通过 rails 命令来创建一个 Rails 应用，例如：

```
rails exam
```

之后将会显示：

```
      create
      create  app/controllers
      create  app/helpers
      create  app/models
      create  app/views/layouts
      create  config/environments
      create  config/initializers
      create  config/locales
```

```
...
```

这样 exam 目录就建好了，下面我们来建一个 metal 应用：

进入 exam 目录，执行：

```
ruby script\generate metal helloservice
```

或

```
ruby script/generate metal helloservice
```

上面的命令取决于你使用的是 Windows 还是 Linux/Unix。

然后我们进入 app 目录下的 metal 目录，编辑 rails 自动生成的 helloservice.rb：

```
# Allow the metal piece to run in isolation
require(File.dirname(__FILE__) + "/../../config/environment") unless defined?(Rails)
require "rubygems"
require 'hproseserver'

def hello(name)
    'Hello ' + name
end

class Helloservice
  def self.call(env)
    if env["PATH_INFO"] =~ /^\/helloservice/
      app = HproseHttpService.new()
      app.add(:hello)
      app.call(env)
    else
      [404, {"Content-Type" => "text/html"}, ["Not Found"]]
    end
  end
end
```

然后回到 exam 目录下，执行：

```
ruby script\server
```

或

```
ruby script/server
```

之后，如果显示类似如下内容：

```
=> Booting Mongrel
=> Rails 2.3.5 application starting on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
```

就说明 Rails 启动成功了。

现在打开浏览器，在地址栏中输入：http://localhost:3000/helloservice/，如果浏览器中显示如下内容：

```
Fa1{s5"hello"}z
```

就说明 Hprose 服务已经发布成功啦。

上面介绍的这几种方式各有优点，大家可以根据实际情况选择适合自己的部署方式。下面我们在介绍 Hprose 具体用法的时候，仍然以 Rackup 方式为例来给大家讲解。

发布服务

现在您已经了解了各种发布服务的方式，下面在来看看如何发布过程、lambda 表达式、语句块和方法吧。

发布函数

Ruby 中实际上没有函数，所有看似函数的定义实际上都是方法。我们这里把属于顶级对象（Object）的方法称为函数。通过 add_function 方法你可以对函数进行发布，例如：

```
app.add_function(:hello)
app.add_function("hello")
app.add_function(method(:hello))
```

这三个写法是一样的，当然你还可以向前面的例子中那样把 add_function 简写成 add，但我们推荐您使用 add_function 这种完整写法，因为它更加清晰也更加高效。

上面三种写法中最保险的是最后一种，因为在 Rackup 方式下，只有最后一种可以工作，因为在 Rackup 方式下，在 ru 文件中定义的方法都不属于顶级对象。

通过 add_function 方法，你还可以发布过程（proc）和 lambda 表达式，例如：

```
app.add_function(proc { |name| 'Hello ' + name + '!' }, "hello")
app.add_function(Proc.new { |name| 'Hello ' + name + '!' }, "hello")
app.add_function(lambda { |name| 'Hello ' + name + '!' }, "hello")
```

上面三种写法是一样的。但是发布过程和 lambda 表达式时，一定要注意最后要跟方法别名。例如下面的写法是错误的：

```
hello = lambda { |name| 'Hello ' + name + '!' }
app.add_function(hello)
```

但是这样写是正确的：

```
hello = lambda { |name| 'Hello ' + name + '!' }
app.add_function(hello, "hello")
```

你也可以给发布的方法提供别名，例如：

```
app.add_function(method(:hello), "hi")
```

但这时，在客户端只能以别名 hi 来调用该方法，而不能用 hello 这个方法名来调用它了。

发布语句块

Hprose for Ruby 提供了一个 add_block 方法专门用来发布语句块，这是一个相当好用的功能，你不需要事先创建一个方法，也不会向发布一个 lambda 表达式或者 proc 那样看上去很复杂。发布语句块的语法看上去相当自然，例如用 add_block 来发布一个 hello 方法是这样写的：

```
app.add_block("hello") { |name|
  'Hello ' + name + '!'
}
```

看上去跟定义一个方法差不多吧。

发布方法

Hprose for Ruby 也支持发布类方法和对象实例方法，如下例：

```
require "rubygems"
require "hproseserver"
class Example1
  def foo
    'foo'
  end
  def bar
    'bar'
  end
  def Example1.foobar
    'foobar'
  end
end
exam1 = Example1.new()
app = HproseHttpService.new()
app.add_methods([:foo, :bar], exam1)
app.add_method(:foobar, Example1)
run app
```

这里 foo，bar 是对象实例方法，它们都属于对象 exam1。而 foobar 是一个类方法，它属于 Example1。

不管是类方法还是实例方法，都可以使用 add_method 一个一个添加，也可以使用 add_methods 成批添加。

另外，前面我们举的例子中：

```
app.add_function(method(:hello))
```

还可以写成：

```
app.add_method(:hello, self)
```

这两种写法是等价的。

现在你可能会有这样的疑问，如果要同时发布两个不同类中的同名方法的话，会不会有冲突呢？如何来避免冲突呢？

别名机制

有时会遇到这种情况，就是发布的方法出现同名时，后添加的方法会将前面添加到方法给覆盖掉，在调用时，你永远不可能调用到先添加的同名方法。不过 Hprose 提供了一种别名机制，可以解决这个问题。要用自然语言来解释这个别名机制的话，不如直接看代码示例更直接一些：

```
require "rubygems"
require "hproseserver"

class Example1
  def foo
    'foo'
  end
  def bar
    'bar'
  end
  def Example1.foobar
    'foobar'
  end
end
exam1 = Example1.new()

class Example2
  def foo
    'foo, too'
  end
  def bar
    'bar, too'
  end
  def Example2.foobar
```

```
    'foobar, too'
  end
end
exam2 = Example2.new()

app = HproseHttpService.new()
app.add_methods([:foo, :bar], exam1, "exam1")
app.add_method(:foobar, Example1, "exam1_foobar")
app.add_methods([:foo, :bar], exam2, "exam2")
app.add_method(:foobar, Example2, "exam2_foobar")
run app
```

从上面这个例子中，我们发现在发布单个方法时，需要指定完整的别名，而在批量发布方法时，只需要添加别名前缀就可以啦。

而实际上，同时添加多个方法（或函数）时，别名也可以是多个，但个数要跟方法（或函数）名个数相同，且一一对应。但一般只指定一个别名前缀即可，别名前缀会跟前面的方法名自动以下划线相连组成别名。

最后要注意的一点是，通过别名发布的方法在调用时如果用原方法名调用是调用不到的，也就是说只能用别名来调用。

发布对象

除了向上面通过 add_method 和 add_methods 发布方法以外，Hprose 可以让您更方便的发布一个对象上的方法，那就是使用 add_instance_methods。add_instance_methods 用来发布指定对象上的指定类层次上声明的所有实例方法。它有三个参数，其中后两个是可选参数。如果您在使用 add_instance_methods 方法时，不指定类层次（或者指定为 nil），则发布这个对象所在类上声明的所有实例方法。这个方法也支持指定名称空间（别名前缀）。

例如上面例子中的：

```
app.add_methods([:foo, :bar], exam1, "exam1")
app.add_methods([:foo, :bar], exam2, "exam2")
```

可以替换为：

```
app.add_instance_methods(exam1, nil, "exam1")
app.add_instance_methods(exam2, nil, "exam2")
```

效果是一样的。

发布类

跟 add_instance_methods 方法类似，使用 add_class_methods 可以让您更方便的发布一个类上的类方法。add_class_methods 有三个参数，其中后两个是可选参数。第一个参数是方法的发布类，第二个参数为方法的执行类，第三个参数为名称空间（别名前缀）。

例如上面例子中的：

```
app.add_method(:foobar, Example1, "exam1_foobar")
app.add_method(:foobar, Example2, "exam2_foobar")
```

可以替换为：

```
app.add_class_methods(Example1, nil, "exam1")
app.add_class_methods(Example2, nil, "exam2")
```

迷失的方法

当客户端调用一个服务器端没有发布的方法时，默认情况下，服务器端会抛出错误。但是如果你希望能对客户端调用的不存在的方法在服务器端做特殊处理的话，你可以通过 add_missing_function 方法来实现。

这是一个很有意思的方法，它用来发布一个特定的方法，当客户端调用的方法在服务器发布的方法中没有查找到时，将调用这个特定的方法。

使用 add_missing_function 发布的方法可以是实例方法、类方法、lambda 表达式或者过程，但是只能发布一个。如果多次调用 add_missing_function 方法，将只有最后一次发布的有效。

用 add_missing_function 发布的方法参数应该为两个：

第一个参数表示客户端调用时指定的方法名，方法名在传入该方法时全部是小写的。

第二个参数表示客户端调用时传入的参数列表。例如客户端如果传入两个参数，则 args 的数组长度为 2，客户端的第一个参数为 args 的第一个元素，第二个参数为 args 的第二个元素。如果客户端调用的方法没有参数，则 args 为长度为 0 的数组。

除了可以直接使用 add_missing_function 来处理迷失的方法以外，你还可以通过 add_function、add_method 或者 add_block 发布一个别名为星号（*）的方法。效果是一样的。

服务器开关

隐藏发布列表

发布列表的作用相当于 Web Service 的 WSDL，与 WSDL 不同的是，Hprose 的发布列表仅包含方法名，而不包含方法参数列表，返回结果类型，调用接口描述，数据类型描述等信息。这是因为 Hprose 是支持弱类型动态语言调用的，因此参数个数，参数类型，结果类型在发布期是不确定的，在调用期才会确定。所以，Hprose 与 Web Service 相比无论是服务的发布还是客户端的调用都更加灵活。

如果您不希望用户直接通过浏览器就可以查看发布列表的话，您可以禁止服务器接收 GET 请求。方法很简单，只需要将 get 属性设置为 false 即可。

好了，现在通过 GET 方式访问不再显示发布列表啦。但是客户端调用仍然可以正常执行，丝毫不受影响。不过在调试期间，不建议禁用发布列表，否则将会给您的调试带来很大的麻烦。也许您更希望能够在调试期得到更多的调试信息，那这个可以做到吗？答案是肯定的，您只要打开调试开关就可以了。

调试开关

默认情况下，在调用过程中，服务器端发生错误时，只返回有限的错误信息。当打开调试开关后，服务器会将错误堆栈信息全部发送给客户端，这样，您在客户端就可以看到详细的错误信息啦。

开启方法很简单，只需要将 debug 属性设置为 true 即可。

P3P 开关

在 Hprose 的 http 服务中还有一个 P3P 开关，这个开关决定是否发送 P3P 的 http 头，这个头的作用是让 IE 允许跨域接收的 Cookie。当您的服务需要在浏览器中被跨域调用，并且希望传递 Cookie 时（例如通过 Cookie 来传递 Session ID），您可以考虑将这个开关打开。否则，无需开启此开关。此开关默认是关闭状态。开启方法与上面的开关类似，只需要将 p3p 属性设置为 true 即可。

跨域开关

Hprose 支持 JavaScript、ActionScript 和 SilverLight 客户端的跨域调用，对于 JavaScript 客户端来说，服务器提供了两种跨域方案，一种是 W3C 标准跨域方案，这个在服务器端只需要将 crossdomain 属性设置为 true 即可。当你在使用 Hprose 专业版提供的服务测试工具 Nepenthes（忘忧草）时，一定要注意必须要打开此开关才能正确进行调试，否则 Nepenthes 将报告错误的服务器。

另一种跨域方案同时适用于以上三种客户端，那就是通过设置跨域策略文件的方式。这个只需要将 crossdomain.xml 放在服务器发布的根目录上即可。对于 Rackup 这种启动服务方式，使用 Rack 的 URLMap 中间件就可以实现。

对于 SilverLight 客户端来说，还支持 clientaccesspolicy.xml 这个客户端访问策略文件，它的设置方法跟 crossdomain.xml 是一样的，都是放在服务器发布的根目录上。

关于 crossdomain.xml 和 clientaccesspolicy.xml 的更多内容请参阅：

- http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html
- http://www.adobe.com/devnet/flashplayer/articles/fplayer9_security.html
- http://msdn.microsoft.com/en-us/library/cc197955%28v=VS.95%29.aspx
- http://msdn.microsoft.com/en-us/library/cc838250%28v=VS.95%29.aspx

服务器事件

也许您可能还希望设置其它的 http 头，或者希望在发生错误时，能够在服务器端进行日志记录。甚至希望在调用发生的前后可以做一些权限检查或日志记录等。在 Hprose 中，这些都可以轻松做到。Hprose 提供了这样的事件机制。

HproseHttpService 提供了四个事件，它们分别是 on_before_invoke、on_after_invoke、on_send_header 和 on_send_error。下面我们就来对这四个事件分别做一下介绍。

on_before_invoke 事件

当服务器端发布的方法被调用前，on_before_invoke 事件被触发，它有四个参数，他们从左到右的顺序分别是 env，name，args 和 byref。其中 env 是环境变量集合，name 为客户端所调用的方法名，args 为方法的参数，byref 表示是否是引用参数传递的调用。

您可以在该事件中做用户身份验证，例如 IP 验证。也可以作日志记录。如果在该事件中想终止调用，抛出异常即可。

on_after_invoke 事件

当服务器发布的方法被成功调用后，on_after_invoke 事件被触发，其中前四个参数与 on_before_invoke 事件一致，最后一个参数 result 表示调用结果。

当调用发生错误时，on_after_invoke 事件将不会被触发。如果在该事件中抛出异常，则调用结果不会被返回，客户端将收到此事件抛出的异常。

on_send_header 事件

当服务器返回响应头部时，on_send_header 事件会被触发，该事件有俩个参数。按从左到右的顺序分别是 env 和 header。其中 env 是环境变量集合，header 是返回的相应头部集合。

在该事件中，您可以发送您自己的头信息，例如设置 Cookie。该事件中不应抛出任何异常。

on_send_error 事件

当服务器端调用发生错误，或者在 on_before_invoke、on_after_invoke 事件中抛出异常时，该事件被触发，该事件有两个参数 env 和 error。其中 env 是环境变量集合，error 是错误对象。

您可以在该事件中作日志记录，但该事件中不应再抛出任何异常。

会话管理

Hprose for Ruby 支持 Session 管理，在创建 HproseHttpService 对象后，将你所使用的 Rack Session 中间件应用于其上即可。

在发布方法中使用 session 很简单，只需要将方法的最后一个参数定义为 session 即可。客户端调用时无需也不应该指定该参数，该参数会自动代入。

Rack 中间件

因为 HproseHttpService 的实例是一个标准的 Rack 应用程序，因此你可以在它之上应用各种 Rack 中间件，例如压缩输出，路径映射等。

例如使用 Rack 中自带一个路径映射的中间件，它的使用方法如下：

```
require "rubygems"
require "hproseserver"

class Example1
  def foo
    'foo'
  end
  def bar
```

```
    'bar'
  end
  def Example1.foobar
    'foobar'
  end
end
exam1 = Example1.new()

class Example2
  def foo
    'foo, too'
  end
  def bar
    'bar, too'
  end
  def Example2.foobar
    'foobar, too'
  end
end
exam2 = Example2.new()

map '/exam1' do
  app = HproseHttpService.new()
  app.add_instance_methods(exam1, nil, "exam1")
  app.add_class_methods(Example1, nil, "exam1")
  run app
end
map '/exam2' do
  app = HproseHttpService.new()
  app.add_instance_methods(exam2, nil, "exam2")
  app.add_class_methods(Example2, nil, "exam2")
  run app
end
```

这样，你就可以使用不同的路径发布不同的服务了，非常方便。

第五章 Hprose for Ruby 客户端

前面我们在快速入门一章里学习了如何创建一个简单的 Hprose for Ruby 客户端，在本章中您将深入的了解 Hprose for Ruby 客户端的更多细节。

本章提要

- 同步调用
- 异步调用
- 异常处理
- HTTP 参数设置

同步调用

直接通过远程方法名进行远程调用

在快速入门一章中，我们已经见识过这种方式的调用了，这里再来具一个例子来进行说明：

```
#!/usr/bin/ruby
require "rubygems"
require "hproseclient"
client = HproseHttpClient.new("http://www.hprose.com/example/")
puts client.sum(1, 2, 3, 4, 5)
puts client.Sum(6.0, 7.0, 8.0)
users = client.getUserList
users.each { |user|
  puts user
  puts "name: " + user.name
  puts "age: " + user.age.to_s
  puts "birthday: " + user.birthday.to_s
  puts "sex: " + user.sex.to_s
  puts "married: " + user.married.to_s
}
```

这个例子的运行结果是：

```
15
21.0
#<User:0x10112ec90>
name: Amy
age: 26
birthday: Sat Dec 03 00:00:00 +0800 1983
sex: 2
married: true
#<User:0x10112efb0>
name: Bob
age: 20
birthday: Mon Jun 12 00:00:00 +0900 1989
sex: 1
married: false
#<User:0x10112eba0>
name: Chris
age: 29
birthday: Sat Mar 08 00:00:00 +0830 1980
sex: 0
married: true
```

```
#<User:0x101128ed0>
name: Alex
age: 27
birthday: Sun Jun 14 00:00:00 +0800 1992
sex: 3
married: false
```

从这个例子中，我们可以看出通过远程方法名进行调用时，远程方法名是不区分大小写的，所以不论是写 sum 还是 Sum 都可以正确调用，如果远程方法返回结果中包含有某个类的对象，而该类并没有在客户端明确定义的话，Hprose 会自动帮你生成这个类的定义（例如上例中的 User 类），并返回这个类的对象。

通过代理对象进行远程调用

如果服务发布方法时使用了别名前缀，那么客户端每次调用时，都需要写带了别名前缀的完整方法名，这样多少有些不便。不过 Hprose 提供了更简单的方式来调用带有别名前缀的方法，那就是通过代理对象。

下面这个例子是通过代理对象来调用“Hprose for Ruby 服务器”→“发布服务”→“别名机制”这一小节中发布的方法：

```
#!/usr/bin/ruby
require "rubygems"
require "hproseclient"
client = HproseHttpClient.new('http://localhost:9292/')
exam1 = client["exam1"]
puts exam1.foo
puts exam1.bar
puts exam1.foobar
exam2 = client[:exam2]
puts exam2.foo
puts exam2.bar
puts exam2.foobar
```

这个例子中，通过 client["exam1"]得到了 exam1 这个代理对象，通过 client[:exam2]得到了 exam2 这个代理对象，之后就可以直接在 exam1 和 exam2 上调用它们的 foo、bar 和 foobar 方法了。

"exam1"和:exam2 在这里作用是一样的。运行结果如下：

```
foo
bar
foobar
foo, too
bar, too
foobar, too
```

另外，如果别名是多个下滑线分割的，例如：my_exam1_foo，可以通过 client["my_exam1"]获取代理对象，也可以 client["my"]["exam1"]分两次来获取代理对象，这是等价的。

通过 Invoke 方法进行远程调用

非引用参数传递

上面介绍的通过远程方法名进行远程调用的例子就是非引用参数传递，下面是用 invoke 方法重写的代码：

```
#!/usr/bin/ruby
require "rubygems"
require "hproseclient"
client = HproseHttpClient.new("http://www.hprose.com/example/")
puts client.invoke("sum", [1, 2, 3, 4, 5])
puts client.invoke("Sum", [6.0, 7.0, 8.0])
users = client.invoke("getUserList")
users.each { |user|
  puts user
  puts "name: " + user.name
  puts "age: " + user.age.to_s
  puts "birthday: " + user.birthday.to_s
  puts "sex: " + user.sex.to_s
  puts "married: " + user.married.to_s
}
```

运行结果与上面例子的运行结果完全相同。但是我们发现用 invoke 方法并不方便，因为当有参数时，必须要把参数单独放入一个数组中才可以进行传递。所以通常我们无需直接使用 invoke 方法，除非我们需要动态调用。另外，还有一种情况下，你会用到 invoke 方法，那就是在进行引用参数传递时。

引用参数传递

下面这个例子很好的说明了如何进行引用参数传递：

```
#!/usr/bin/ruby
require "rubygems"
require "hproseclient"
client = HproseHttpClient.new("http://www.hprose.com/example/")
args = [{"Mon"=>1, "Tue"=>2, "Wed"=>3, "Thu"=>4, "Fri"=>5, "Sat"=>6, "Sun"=>7}]
puts args
result = client.invoke("swapKeyAndValue", args, true)
puts args
```

上面程序的运行结果如下：

```
Wed3Sun7Thu4Mon1Tue2Sat6Fri5
5Fri6Sat1Mon7Sun2Tue3Wed4Thu
```

我们看到运行前后，args 中的值已经改变了。

这里有一点要注意，当参数本身是 Hash 或数组时，该 Hash 或数组应该作为参数数组的第一个元素传递，例如上例中的 args 是：

```
[{"Mon"=>1, "Tue"=>2, "Wed"=>3, "Thu"=>4, "Fri"=>5, "Sat"=>6, "Sun"=>7}]
```

而不能只写：

```
{"Mon"=>1, "Tue"=>2, "Wed"=>3, "Thu"=>4, "Fri"=>5, "Sat"=>6, "Sun"=>7}
```

否则程序将会出错，或者在调用中陷入等待状态，这样的错误有时候不容易被找到，因此一定要注意这一点。

异步调用

非引用参数传递

异步调用是通过指定回调函数的方式来实现的，看下面这个例子：

```
#!/usr/bin/ruby
require "rubygems"
require "hproseclient"
client = HproseHttpClient.new("http://www.hprose.com/example/")
client.sum(1, 2, 3, 4, 5) { |result|
  puts result
}
client.Sum(6.0, 7.0, 8.0) { |result, args|
  puts result
  puts args
}
client.getUserList { |users|
  users.each { |user|
    puts user
    puts "name: " + user.name
    puts "age: " + user.age.to_s
    puts "birthday: " + user.birthday.to_s
    puts "sex: " + user.sex.to_s
    puts "married: " + user.married.to_s
  }
}
gets
```

它与前面同步调用的第一个例子是相同的，但是结果顺序不一定是按照调用的顺序输出的。

回调方法由语句块构成，参数个数可以是 0 个，也可以是 1 个或 2 个。如果参数个数是零个，在执行

回调方法时，不会有结果传入，这通常用于调用没有结果的远程方法。如果参数个数是一个，在执行回调方法时，结果会作为参数来传入。如果参数个数是两个，则第一个参数是执行结果，第二个参数是调用参数。

引用参数传递

跟同步方式相同，在异步方式下，只能通过 invoke 方式进行引用参数传递的远程调用。例如：

```
#!/usr/bin/ruby
require "rubygems"
require "hproseclient"
client = HproseHttpClient.new("http://www.hprose.com/example/")
args = [{"Mon"=>1, "Tue"=>2, "Wed"=>3, "Thu"=>4, "Fri"=>5, "Sat"=>6, "Sun"=>7}]
puts args
result = client.invoke("swapKeyAndValue", args, true) { |result, args|
  puts args
}
gets
```

这个运行结果为：

```
Wed3Sun7Thu4Mon1Tue2Sat6Fri5
5Fri6Sat1Mon7Sun2Tue3Wed4Thu
```

同样，异步调用也支持通过代理对象进行远程调用，用法跟上面类似，结合同步调用和上面异步调用的例子，您应该可以做到无师自通了。

异常处理

同步调用中的异常处理

Hprose for Ruby 的客户端在同步调用过程中，如果服务器端发生错误，或者客户端本身发生错误，异常将在客户端被直接抛出，使用 begin...rescue 语句块即可捕获异常，通常服务器端调用返回的异常是 HproseException 类型。但是在调用过程中也可能抛出其它类型的异常。

例如，当调用不存在的方法时：

```
#!/usr/bin/ruby
require "rubygems"
require "hproseclient"
client = HproseHttpClient.new("http://www.hprose.com/example/")
begin
  client.notexisted
rescue ::Exception => e
  puts e
```

```
end
```

运行结果如下：

```
Can't find this function notexisted().
file: /var/www/hprfc/website/example/hproseHttpServer.php
line: 157
trace: #0 /var/www/hprfc/website/example/hproseHttpServer.php(385): HproseHttpServer->doInvoke()
#1 /var/www/hprfc/website/example/hproseHttpServer.php(402): HproseHttpServer->handle()
#2 /var/www/hprfc/website/example/index.php(60): HproseHttpServer->start()
#3 {main}
```

异步调用中的异常处理

Hprose for Ruby 的客户端在异步调用过程中，如果服务器或客户端发生错误，异常不会被抛出，不会影响程序执行，但如果需要处理异常，可以通过 onerror 事件来捕捉所有异步调用过程中发生的异常，例如：

```
#!/usr/bin/ruby
require "rubygems"
require "hproseclient"
client = HproseHttpClient.new("http://www.hprose.com/example/")
client.onerror = lambda { |name, error|
  puts name
  puts error
}
client.notexisted {}
gets
```

上面这个程序中，我们通过 lambda 表达式定义了 onerror 事件，它有两个参数，第一个参数表示发生错误时所调用的方法名，第二个参数是捕捉到的异常。上面的程序运行结果如下：

```
notexisted
Can't find this function notexisted().
file: /var/www/hprfc/website/example/hproseHttpServer.php
line: 157
trace: #0 /var/www/hprfc/website/example/hproseHttpServer.php(385): HproseHttpServer->doInvoke()
#1 /var/www/hprfc/website/example/hproseHttpServer.php(402): HproseHttpServer->handle()
#2 /var/www/hprfc/website/example/index.php(60): HproseHttpServer->start()
#3 {main}
```

注意：onrrror 事件只对异步调用有效，对同步调用无效。

HTTP 参数设置

目前的版本只提供了 http 客户端实现，针对于 http 客户端，有一些特别的设置，例如代理服务器、http 标头等设置，下面我们来分别介绍。

代理服务器

默认情况下，代理服务器是被禁用的。可以通过 proxy 属性来设置 http 代理服务器的地址和端口，它是一个只写属性。参数是代理服务器的完整地址字符串，例如："http://10.54.1.39:8000"。

HTTP 标头

有时候您可能需要设置特殊的 http 标头，例如当您的服务器需要 Basic 认证的时候，您就需要提供一个 Authorization 标头。设置标头很简单，只需要设置 header 属性就可以啦，header 是一个 Hash 类型的属性，虽然它本身是只读的，但是你可以设置它的元素。key 表示标头名，value 表示标头值，key 和 value 都是字符串型。

标头名不可以为以下值：

- Context-Type
- Context-Length
- Host

因为这些标头有特别意义，客户端会自动设定这些值。

另外，Cookie 这个标头不要轻易去设置它，因为设置它会影响 Cookie 的自动处理，如果您的通讯中用到了 Session，通过 header 属性来设置 Cookie 标头，将会影响 Session 的正常工作。